**Similar, yet Different, Concepts**

- Inner classes
- Inheritance
- Both are based on hierarchical ("tree-like") structures

**Inner Classes**

- In Java (and C++), classes can be nested within one another.
- Objects in the inner class has available instance variables and methods of the outer class.

**Ways to Construct ClosedList**

- `class Cell {...}
class ClosedList {...}

- `class ClosedList {
class Cell {...}
...
}`

**Interpretation of Identifiers**

- In an inner class, the innermost meaning of an identifier applies.

```
class ClosedList {
    String identity;
class Cell {
    String identity;
    ...
} ...
```

**Usage**

- Normally one or more objects of the inner class are created for a given object of the outer class.
- Objects of the inner class only make sense in the context of a supporting object of the outer class.

**Exporting Inner Objects**

- Inner objects can be used outside, understanding that they are always relative to the object in which they are contained.
Example: List Iterator

- We want to define an Iterator for a CloseList.
- For read-only iteration, the Iterator class can be defined outside the CloseList class.
- For modification, such as remove(), it is sometimes necessary to change part of the list itself.

ClosedList.Iterator

```java
class ClosedList {
    private Cell head;
    private Cell tail;
    ...
    public Iterator getIterator() {
        return new Iterator(head);
    }
    public class Iterator // inner class to ClosedList {
        private Cell current;
        private Cell previous;
        public Iterator(Cell head) {
            current = head;
            previous = null;
        }
        ...
    }
}
```

Aside: Converting a list to a String (e.g. for printing entire list at once)

```java
public String toString() {
    StringBuffer buffer = new StringBuffer();
    Iterator it = new Iterator(head);
    if (it.hasNext()) {
        buffer.append(it.next()); // first element
    }
    while (it.hasNext()) {
        buffer.append(" "+ it.next()); // leave space
    }
    return buffer.toString();
}
```

Test Program

```java
class ClosedListTest {
    public static void main(String arg[]) {
        int numItems = 10;
        ClosedList L = new ClosedList();
        for (int i = 0; i < numItems; i++) {
            L.enqueue(new Integer(i));
        }
        ClosedList.Iterator it = L.getIterator();
        System.out.println("removing " + it.next());
        it.remove(); // remove first item
    }
}
```
```
public class ClosedListTest {
    public static void main(String args[]) {
        int numItems = 10;
        ClosedList L = new ClosedList();
        // add 10 items to L
        for (int i = 0; i < numItems; i++) {
            L.enqueue(new Integer(i));
        }
        System.out.println("Initial list contents: " + L);
        // starting from the beginning, skip 3 items
        ClosedList.Iterator it = L.getIterator();
        for (int i = 0; i < 3; i++) {
            System.out.println("skipping " + it.next());
        }
        // remove 2 items
        System.out.println("Removing " + it.next());
        it.remove();
        System.out.println("Removing " + it.next());
        it.remove();
        System.out.println("List contents after removing two: " + L);
        // insert 3 items
        for (int i = 0; i < 3; i++) {
            int value = 10 * (i + 1);
            System.out.println("Inserting " + value);
            it.insert(new Integer(value));
        }
        System.out.println("List contents after inserting three: " + L);
    }
}
```

Test Program

Test Program Output

```
Initial list contents: 0 1 2 3 4 5 6 7 8 9
skipping 0
skipping 1
skipping 2
removing 3
removing 4
List contents after removing two: 0 1 2 5 6 7 8 9
skipping 5
skipping 6
inserting 10
inserting 20
inserting 30
List contents after inserting three: 0 1 2 5 6 7 10 20 30 8 9
```

Inheritance

"Inheritance" is a way of building one class on top of another. The original class is called the base class, or parent class. The new class is called the derived class, or child class.

Additional Aspects of Object-Oriented Programming:

Inheritance

Diagrammatic Notation (UML)

Inherited Capabilities

- **Extension:**
  The derived class can potentially use all data components and methods from the base class, and add more of its own.

- **Over-Riding:**
  It can also selectively re-define or "over-ride" methods of the same name.

```
class Professor extends Employee {
    public void teach() {
        System.out.println("Teaching...");
    }
}
```
Purposes of Inheritance

- Use the same concepts and code for many classes (base-class concepts and code shared by derived classes):
  - Work economy
  - Intellectual economy
- Tie together similar classes:
  - Increases the utility of methods that use such classes.

Extension = Java Inheritance

- In Java, the keyword for "inherits from" is extends
- The derived class extends the base class.
- Extension allows over-riding as well; there is no separate keyword for over-riding.

Extension Example

- class Account defines a basic bank account
- class CheckingAccount defines a special account for check-writing

```java
class Account {
    Money balance;
    Account(Money initialBalance) {
        balance = initialBalance;
    }
    void deposit(Money amount) {
        balance = balance.add(amount);
    }
    boolean withdraw(Money amount) {
        if (balance.lessThan(amount)) return false;
        balance = balance.subtract(amount);
        return true;
    }
    void showBalance(PrintStream out) {
        out.println("Balance: "+ balance);
    }
}

class CheckingAccount extends Account {
    Money serviceCharge;
    CheckingAccount(Money initialBalance, Money serviceCharge) {
        super(initialBalance);
        this.serviceCharge = serviceCharge;
    }
    boolean cashCheck(Money amount) {
        return withdraw(amount.add(serviceCharge));
    }
}
```

(continued next page)
Multiple derived classes

Over-Ride Example: Inheriting from Applet

```java
import java.applet.*;

public class MyApplet2 extends Applet {
    public void update(Graphics g) {
        paint(g);
        update();
    }
}
```

When any client calls `update()`, this is the one that gets called.

Which methods can be over-ridden?

- In order to be over-ridden, a method must be declared either:
  - `public`
  - `protected`
  - in the base class.

Implications of Inheritance

- The preceding diagram means, for example, that to find all methods for class `java.util.Stack`, you may wish to look at:
  - `java.util.Vector`
  - `java.util.AbstractList`
  - `java.util.AbstractCollection`
  - `java.lang.Object`

Methods of `java.util.Stack`

- Methods of `Stack` proper:
  - `toString`, `hashCode`, `equals`, `notify`, `notifyAll`, `wait`
- Methods of `Vector`:
  - `add`, `add`, `addAll`, `removeAll`, `removeAllElements`, `removeElement`, `removeElementAt`, `removeRange`, `retainAll`, `set`, `setElementAt`, `setSize`, `toArray`, `toString`
- Methods of AbstractList:
  - `iterator`, `listIterator`
- Methods of AbstractCollection:
  - `add`, `remove`, `addAll`, `removeAll`
Testing where Object is in Hierarchy

- `instanceof` operator

```java
Object ob = ...;
if (ob instanceof Vector) ...
if (ob instanceof Stack) ...
```

Casting

```
Object
  ↑
  |  up-casting
  |  always safe
  ↓
Vector
  |
  ↓
Stack
```

If you down-cast and you are wrong, you will get a `ClassCastException`, which can terminate your program.

class `Object`

- `Object` is the ancestor of all classes
- Some methods of `Object`:
  - `boolean equals(Object)`
  - `Class getClass(): returns a Class object`
  - `String toString()`
- Method of class `Class`:
  - `String getName()`

Mouse-Handing, e.g. in Applets

- Two ways, both based on things we have seen:
  - `over-ride` methods in base class
  - define handlers as `inner classes` and attach them
- The first method is deprecated in current Java, but still useable.

Handling by Over-Riding

- Applet declares methods:
  - `mouseDown`
  - `mouseUp`
  - `mouseDrag`
- Each method is called automatically when the corresponding event occurs.
- Derived class over-rides these.

Example: `EZdraw.java` (extends `Applet`)

```java
/* remember the current mouse coordinates */
private void remember(int x, int y)
{
  last_x = x; last_y = y;
}

/** mouseDown is called when user presses the mouse to start a drawing */
public boolean mouseDown(Event e, int x, int y)
{
  remember(x, y);
  return true;
}

/** mouseDrag is called when user drags the mouse */
public boolean mouseDrag(Event e, int x, int y)
{
  int[] last = new int[2];
  Graphics g = getGraphics();
  g.setColor(last_x, last_y, x, y);
  g.drawLine(last_x, last_y, x, y);
  return true;
}
```

These must be declared with the same type and visibility as in base class.
Handling by Inner Classes

- Applet defines inner classes for various types of handling.
- Applet adds handlers at initialization.

Example: SoSodraw.java (extends Applet)

```java
public void init()
{
    ...;
    addMouseListener(new SoSoMouseListener());
    addMouseMotionListener(new SoSoMouseMotionListener());
}

class SoSoMouseListener implements MouseListener
{
    /* mousePressed is called when user presses the mouse */
    public void mousePressed(MouseEvent e)
    {
        int x = e.getX();
        int y = e.getY();
        remember(x, y);
    }

    Must also define, even if you don’t use them:
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

Example: SoSodraw.java (2)

```java
class SoSoMouseMotionListener implements MouseMotionListener
{
    /** mouseDragged is called when user drags the mouse */
    public void mouseDragged(MouseEvent e)
    {
        int x = e.getX();
        int y = e.getY();
        Graphics g = getGraphics();
        g.drawLine(last_x, last_y, x, y);
        remember(x, y);
    }

    public void mouseMoved(MouseEvent e)
    {
    }
}
```

Implementing an Interface is similar to Inheritance

- Interface ≈ Base Class
- Implementor ≈ Derived Class
- By declaring methods to use the Interface rather than the Implementor class as an argument, more generality is afforded to that method.

Example

- `java.util.Iterator` is standard
- Make `ClosedList.Iterator` implement `java.util.Iterator`
- Any other code accepting a `java.util.Iterator` can now use our `ClosedList.Iterator`
- We can still do everything we did before.

Morals

- When possible, write your methods to use Interfaces rather than classes.
- Define Interfaces for your classes, on which others can depend.
- Ideally, define the Interfaces first.